

SYSTEM AND METHOD FOR CONTEXT-BASED SERIALIZATION OF MESSAGES IN A PARALLEL EXECUTION ENVIRONMENT

Field of the Invention

{001} The present invention relates generally to messaging, and more particularly relates to messaging in a parallel execution environment.

Background

{002} Messaging, as discussed here, may be defined as a method that allows two entities to communicate by sending and receiving messages without requiring human interaction. One of the most important aspects of messaging is its asynchronous nature: the sender of the message does not need to wait for the recipient to receive the information. Sending applications are free to generate messages at an appropriate speed, handling peak periods as they occur, without having to wait for recipients to deal with the requests.

{003} Messaging methods vary in their implementations. Two of the most common implementations are the hub-and-spoke architecture and the bus architecture.

{004} In the hub-and-spoke architecture, applications are connected to a central processor, which may be called a message server. The message server handles all communication among the connected applications, which are called the application clients. An application client can be a sender of a message, a recipient, or both.

{005} The message server is responsible for routing messages correctly, authenticating and authorizing user access, and guaranteeing the delivery of messages.

{006} A bus architecture does not have a centralized message server to coordinate the distribution of messages. Instead, each application client includes the functionality typically found in a message server. Application clients are connected to a message bus, which typically uses the network layer of the IP multicast protocol. Although the multicast network layer routes messages among each of the application clients, the clients must perform all checks and balances to ensure that the messages are delivered securely and reliably.

{007} One of the major problems of today's messaging methods concerns the execution of related requests in a parallel execution environment. In FIG. 1, clients 10A-10N send messages to execution systems 17A and 17B through a message queue 14. For example, a client sends a payment message request to a bank. If the client sends another related request immediately following the payment request, e.g., to modify or cancel the payment request, the messaging method must always preserve the order of the original sequence of requests to guarantee a consistent execution. In a sequential execution environment there is no problem, since all requests are executed according to their original sequence.

{008} In a parallel execution environment, however, e.g., an environment where the target system has two or more processors, preservation of the context-based sequence cannot be guaranteed. The parallel execution system starts a new thread for each incoming message request and executes requests without regard to their intended order. This may transpose the execution of two consecutive requests that are related. For example, a request to cancel a payment request may be executed before the payment request itself. Here, the request to cancel the payment request

fails because there is nothing to cancel at the time the cancellation is executed, and the payment subsequently goes through despite its intended cancellation.

{009} Existing messaging systems, for example the IBM MQ Series, do not provide any functionality for processing context-based requests in parallel execution environments without human interaction.

{010} Thus, there is a need to provide an improved messaging system and method which is applicable in a parallel execution environment and which guarantees the automatic execution of related requests according their context-based or original sequence, without requiring human interaction.

Summary

{011} The present invention includes an improved messaging system and method which allows parallel execution of related requests according their context-based sequence by using a serialization processor. The serialization processor receives each incoming message request from the messaging client, retrieves a transaction identifier (TI), and checks a state table for the retrieved TI. When the TI is found already stored in the state table with the state "active," the serialization processor stores the new requests with the same TI in a serialization queue and makes an new entry for that TI with the state "queued" in the state table. Once an active request has been executed, its entry in the state table is cleared. The queued request with the same TI is executed by the execution system, and its entry is updated from the state "queued" to the state "active." In a preferred embodiment of the present invention, the headers of message requests may contain message identifiers (MI), which mark the message requests as context-based-serial. Message requests that do not have a message identifier are executed without involvement of the serialization processor.

{012} These and additional features and advantages of the present invention will become more apparent from the detailed description set forth below when taken in conjunction with the drawings.

Brief Description of the Drawings

{013} The accompanying drawings illustrate embodiments of the present invention and, together with the description, further serve to explain the principles of the embodiments of the present invention.

{014} FIG. 1 shows an example of the messaging system having a parallel execution environment according to the prior art.

{015} FIG. 2 shows incorporation of the present invention into a parallel execution environment.

{016} FIG. 3 shows a typical hub-and-spoke messaging system where multiple clients put their messages into a message queue.

{017} FIG. 4 shows a typical message suitable for use by the inventive serialization processor.

{018} FIG. 5 shows an example of a message header of the message system of the IBM MQ Series.

{019} FIG. 6 shows a typical message queue layout suitable for use by the inventive serialization processor.

{020} FIG. 7 shows a preferred structure of the state table as used by the inventive serialization processor.

{021} FIG. 8 shows a flowchart illustrating steps of a method performed by the inventive serialization processor.

Detailed Description

{022} The following hardware and software provide a suitable context for implementation of the present invention: a messaging client installed at the client side, for example the IBM MQ Series Client; a network connection to the server system; and an operating system depending on the clients, for example Microsoft Window 2000 for PC-based clients. At the server side a messaging server is needed, for example the IBM MQ Series Server. A state table as proposed by the present invention may be implemented using a data base management system such as the IBM DB2. The server should have a multi-tasking operating system like Windows 2000.

{023} As shown in FIG. 3, clients 10A-10N put their message requests 12 into a message queue 14 which is managed by a queue manager 15. If the message header of an incoming message defines that message as persistent, the message is then stored persistently; otherwise it is stored non-persistently. Non-persistent messages reside in the memory only, and may get lost if the queue manager 15 crashes. Persistent messages are stored on a persistent medium, for example a disk, and cannot be inadvertently lost. Each of the clients 10A-10N includes a client interface layer in order to communicate with the queue manager 15 at the server side.

{024} The inventive serialization processor 16 gets each message 12 from the message queue 14, and analyzes the message to find the transaction identifier (TI) if the message context indicates that the message has a TI. The transaction identifier may be defined as an identifier in a message which identifies all messages belonging to the same business transaction. Then the serialization processor 16 searches a state table 18, which is preferably realized as a table in a database system. The state table 18 provides information regarding which messages are “queued” or “active” (being executed). The query returns two possible results: either the TI cannot be found in the state table 18, or the TI is found in the state table 18 with the state “active.”

{025} When the TI cannot be found in the state table 18, the serialization processor 16 creates a new entry in the state table 18 that contains the new TI and the state “active.” Then the serialization processor 16 puts the message into one of the execution queues 19A, 19B shown in FIG. 2. The message is then executed by one of the threads of the connected execution system 17A, 17B. When the thread is finished processing but before it closes, the TI is cleared in the state table 18.

{026} When the TI is found in the state table 18 with the state “active,” the new message will be put into a serialization queue 20. In the serialization queue 20, the message is held until the active thread has finished and the TI has cleared in the state table 18. If a further message that has the same TI arrives at the message queue 14, the newly arrived message will be put into the serialization queue 20.

{027} From time to time, the serialization processor 16 checks for messages held in the serialization queue 20. A number of different mechanisms may trigger this check in different embodiments. For example, a timer interrupt may cause the serialization processor 16 to check for messages in the serialization queue 20. In

another embodiment, the executing thread may set a flag to inform the serialization processor 16 that it has finished its task and has cleared the TI in the state table 18. If the serialization processor 16 is triggered by such an event, it checks the serialization queue 20 for messages, and if a message is found, determines whether the TI has already been cleared from the state table 18. If the TI has been cleared, the message is removed from the serialization queue 20 and put into one of the execution queues 19A, 19B for processing.

{028} The present invention may also be implemented as follows: When the transaction identifier is found in the state table 18 with the state “active,” the new message may be put into the serialization queue 20, and a new entry created for that message in the state table 18 with the status “queued.” After a defined time interval, the serialization processor 16 checks the state table 18, and changes the state to “active” once the related messages with the entry “active” have been cleared.

{029} FIG. 3 shows a typical hub-and-spoke messaging system where multiple clients 10A-10N put their messages into a message queue 14. Each of the clients 10A-10N has a client interface 11A-11N with the queue manager 15. Furthermore, the inventive serialization processor 16 (which, in the interest of simplicity, is not shown in FIG. 3) has an interface with the queue manager 15 as well as with the application system. The serialization processor 16 ensures that an application thread is assigned to a message only when the serialization processor 16 puts the message into one of the execution queues 19A, 19B.

{030} FIG. 4 shows a typical message layout suitable for use by the serialization processor 16. The message 12 includes a message header 24 and a message body 26. The message header 24 normally includes a message identifier, a source queue name, a target queue manager, the target queue name, the message type, and a code

page used for the message body 26. Messages lacking a context-based identifier are treated as unrelated, and do not need to be executed by the serialization processor 16. The message body 26 normally contains the transaction identifier (TI) and the user (application) defined data.

{031} The message body 26 is handled primarily by the execution or application system rather than the messaging system. Consequently, a traditional message system itself cannot react to a TI that is part of the message body 26. Therefore, the TI is inspected by the serialization processor 16 in order to avoid parallel execution for two messages which belong to the same TI. FIG. 5 shows an exemplary message header of the IBM MQ Series message system, which may be used by the present invention.

{032} FIG. 6 shows preferred embodiment of the message queue 14 as used by the present invention. Messages are put into the message queue 14 by the clients 10A-10N. On the right-hand side of FIG. 6, the server side, messages are taken out by the serialization processor 16. In the inventive messaging system as described here, each message taken out of the message queue 14 creates a new application thread which handles the body of the message. There may be multiple server machines connected to the same queue, taking the messages in parallel.

{033} FIG. 7 shows a preferred embodiment of the state table 18 as used by the present invention. Each incoming new message request contains a transaction identifier (TI) 28 in its message body. The serialization processor 16 checks the state table 18 to determine whether this transaction identifier 28 is already stored, and whether the transaction state 30 is active. If both conditions are satisfied, the transaction identifier 28, the state 30 "queued," and the message ID 33 taken from the message header 24 are stored in the state table 18 together with a time stamp 36 recording the arrival time of the new message. The state table 18 may preferably

be implemented as a persistent table in a data base system, with columns for the transaction ID, message ID, and time stamp. This ensures that messages will not be lost.

{034} If an incoming message request contains a TI which is not found in the state table 18, the new TI is stored in the state table 18 with the state 30 “active,” together with its arrival time (time stamp 36) and the message ID 33 from the message header. Then the message is delivered to an application thread for processing. When the tread has completed its processing, the entry is cleared from the state table 18.

{035} The possible transaction states 30 in the state table 18 are active and queued. The state “active” indicates that another message with the same TI is currently being processed by one of the application threads. The state “queued” indicates that another message with the same TI is already queued, and is waiting for processing, in which case the new message will also be queued.

{036} The message ID 33 may also be stored in the state table 18, and is needed in order to retrieve the correct message from the serialization queue 20 when the processing of the active transaction is finished.

{037} Furthermore, the time stamp 36 of the arrival time of the message is needed when more than one entry with the same TI is found in the state table 18. In this case, the saved messages must be put into the application queues according to the arrival sequence.

{038} To ensure that requests are not lost, a logical unit of work (LUW) must span all activities necessary to process a message. A LUW will be started when a message is received from the message queue 14. Then, the data base holding the

state table 18 is accessed to find the transaction identifier 28, after the message is put into one of the execution queues 19A, 19B. The state table 18 is updated to reflect the new transaction identifier 28. Then a commit operation must be issued which commits all activities. This guarantees that the present message is deleted from the message queue 14, that the message is available for processing in the execution queue 19A or 19B, and that the transaction identifier 28 is stored in the state table 18. The same transaction processing must be performed when a saved message is taken out of the serialization queue 20 and sent to one of the execution queues 19A, 19B for processing.

{039} FIG. 8 shows a flowchart of a process according to the inventive method. The client application may be, for example, a payment application. The user creates a payment message which includes the following data: the amount of money to be paid, the sender, the receiver, and a transaction identifier. The payment application creates, using the message client, a payment message (step100). The payment message contains a message header and message body as mentioned earlier with reference to FIG. 5. The payment message is sent to the message server by means of the message client and the network (step 150).

{040} The message server receives the payment message and puts the payment message into the message queue 14 using the queue manager 15 (step 200). The queue manager 15 stores the payment message persistently or non-persistently, depending on the data in the message header. The serialization processor 16 takes the payment message from the message queue (step 250), analyses the payment message for the transaction identifier (step 300), and searches the state table 18 for the same transaction identifier (step 350).

{041} If the transaction identifier is not already contained in the state table 18 (step 400), the serialization processor 16 creates a new entry in the state table 18

with the transaction identifier of the new message, its message ID, its time stamp, and its state “active” (step 500), and sends the payment request to the execution system for processing of the payment message (step 600). After the payment message is executed by the execution system or application, the entry in the state table 18 is cleared (step 700).

{042} Otherwise (i.e., the transaction identifier is already stored in the state table 18) (step 450), the serialization processor 16 puts the new payment message into the serialization queue 20 (step 550) and, once the same transaction identifier has been cleared in the state table 18, the serialization processor 16 sends the payment message to the execution system for processing (step 650).

{043} In another embodiment of the present invention, the new message is sent to the serialization queue 20, and a new entry for the new message is created in the state table 18 with the status “queued.” After a defined time interval, the serialization processor 16 checks the state table 18 and changes the state “queued” to the state “active” once the related message with the entry “active” has been cleared.